TECH SHOP

Implement OOPs

Task 1: Classes and Their Attributes:

Task 2: Class Creation:

• Create the classes (Customers, Products, Orders, OrderDetails and Inventory) with the specified attributes.

• Implement the constructor for each class to initialize its attributes.

• Implement methods as specified.

Task 3: Encapsulation:

• Implement encapsulation by making the attributes private and providing public properties (getters and setters) for each attribute.

• Add data validation logic to setter methods (e.g., ensure that prices are non-negative, quantities are positive integers).

Customers Class:

Attributes:

• CustomerID (int)

• FirstName (string)

• LastName (string)

• Email (string)

• Phone (string)

• Address (string)

Methods:

• CalculateTotalOrders(): Calculates the total number of orders placed by this customer.

• GetCustomerDetails(): Retrieves and displays detailed information about the customer.

• UpdateCustomerInfo(): Allows the customer to update their information (e.g., email, phone, or address).

InventoryDao.py | OrderDetailsDao.py | OrderDao.py | Customer.py

TechShop > entity > Customer.py > Customer > update_customer_info

```python
import re
from exception.exceptions import InvalidDataException


class Customer:
    def __init__(self, customer_id=None, first_name="", last_name="", email="", phone="", address=""):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email = self.__validate_email(email)
        self.__phone = phone
        self.__address = address
        self.orders = []

    def __validate_email(self, email):
        if re.match(r"[^@]+@[^@]+\.[^@]+", email):
            return email
        else:
            raise InvalidDataException("Invalid email address provided.")

    def calculate_total_orders(self):
        return len(self.orders)

    def get_customer_details(self):
        return f"Customer ID: {self.__customer_id}\nName: {self.__first_name} {self.__last_name}\nEmail:

    def update_customer_info(self, email=None, phone=None, address=None):
        if email:
            self.email = email
        if phone:
            self.phone = phone
        if address:
            self.address = address

    @property
    def customer_id(self):
        return self.__customer_id
```

```
TechShop > entity > Customer.py > Customer > update_customer_info
class Customer:

    @property
    def first_name(self):
        return self.__first_name

    @first_name.setter
    def first_name(self, first_name):
        if isinstance(first_name, str):
            self.__first_name = first_name
        else:
            raise InvalidDataException("First name must be a string.")

    @property
    def last_name(self):
        return self.__last_name

    @last_name.setter
    def last_name(self, last_name):
        if isinstance(last_name, str):
            self.__last_name = last_name
        else:
            raise InvalidDataException("Last name must be a string.")

    @property
    def email(self):
        return self.__email

    @email.setter
    def email(self, email):
        self.__email = self.__validate_email(email)

    @property
    def phone(self):
        return self.__phone

    @phone.setter
    def phone(self, phone):
        # Phone number validation: must be a string containing only digits
        if isinstance(phone, str) and phone.isdigit():
            self.__phone = phone
        else:
            raise InvalidDataException("Phone number must be a string containing only digits.")

    @property
    def address(self):
        return self.__address

    @address.setter
    def address(self, address):
        if isinstance(address, str):
            self.__address = address
        else:
            raise InvalidDataException("Address must be a string.")
```

Products Class:

Attributes:

• ProductID (int)

• ProductName (string)

• Description (string)

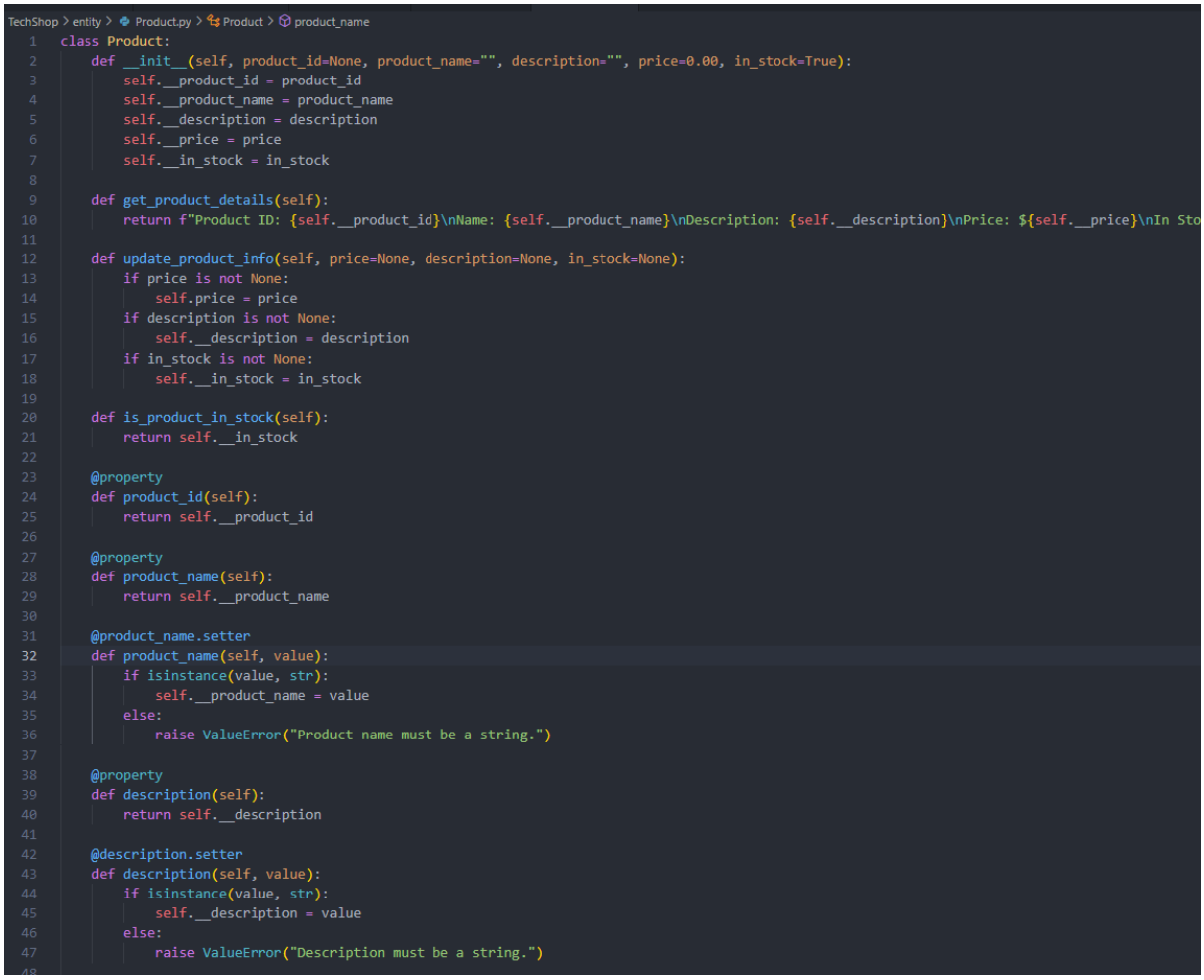
• Price (decimal)

Methods:

• GetProductDetails(): Retrieves and displays detailed information about the product.

• UpdateProductInfo(): Allows updates to product details (e.g., price, description).

• IsProductInStock(): Checks if the product is currently in stock.



```python
    @property
    def price(self):
        return self.__price

    @price.setter
    def price(self, value):
        if isinstance(value, (int, float)):
            if value >= 0:  # Check if price is non-negative
                self.__price = value
            else:
                raise ValueError("Price must not be negative.")
        else:
            raise ValueError("Price must be a numeric value.")
```

Orders Class:

Attributes:

• OrderID (int)

• Customer (Customer) - Use composition to reference the Customer who placed the order.

• OrderDate (DateTime)

• TotalAmount (decimal)

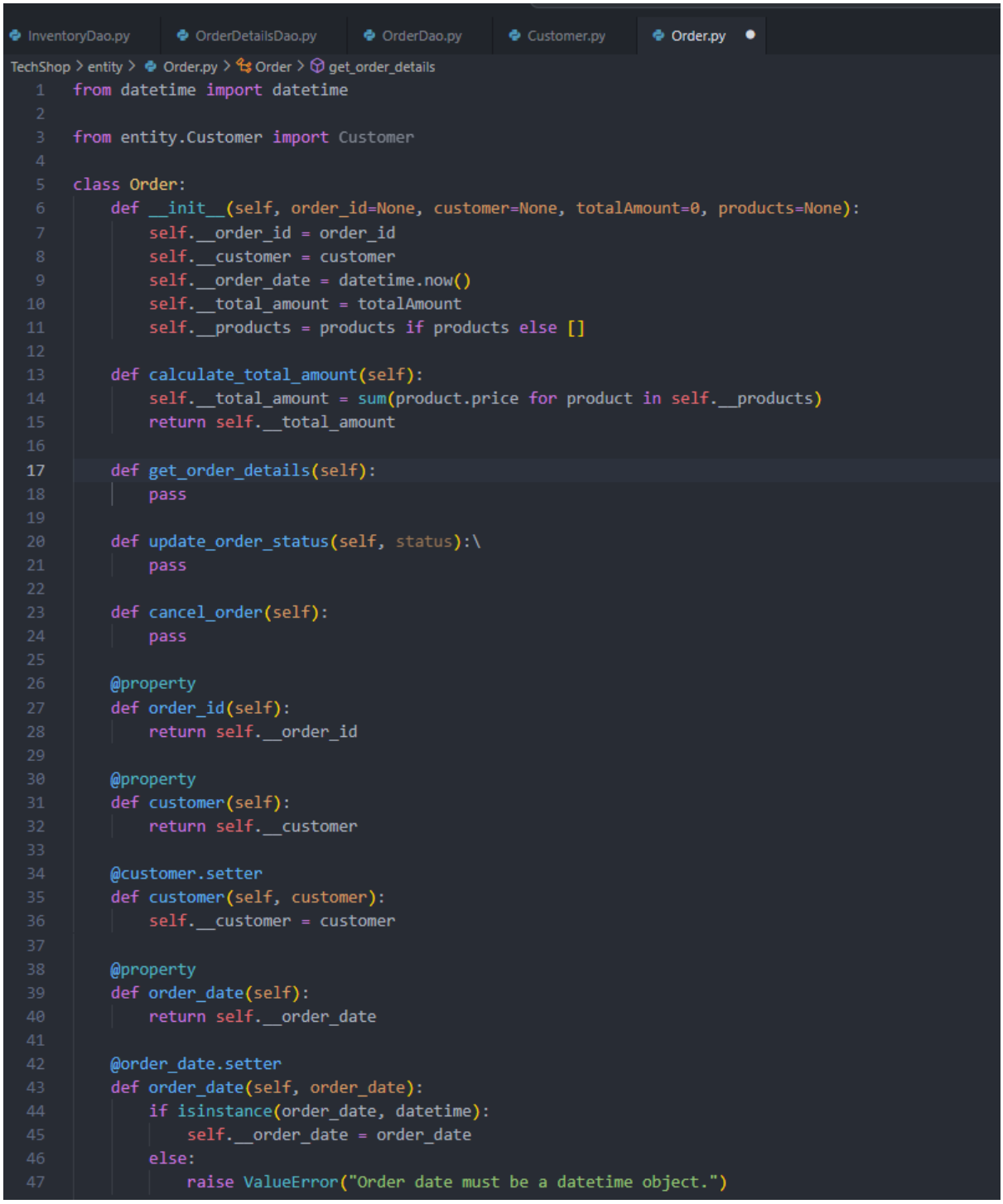
Methods:

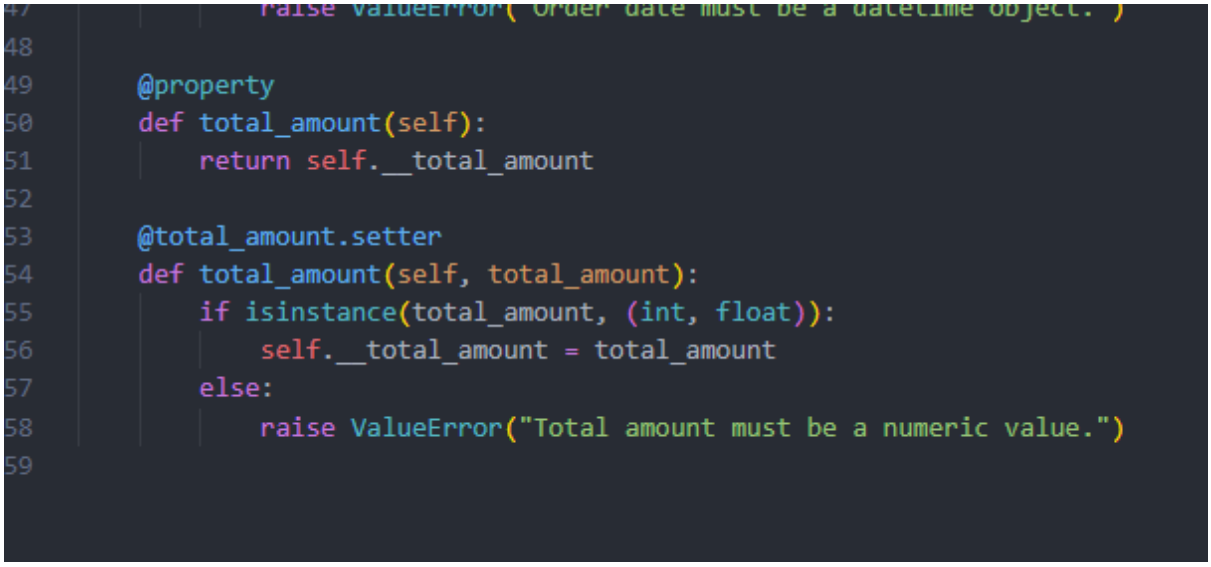
• CalculateTotalAmount() - Calculate the total amount of the order.

• GetOrderDetails(): Retrieves and displays the details of the order (e.g., product list and quantities).

• UpdateOrderStatus(): Allows updating the status of the order (e.g., processing, shipped).

• CancelOrder(): Cancels the order and adjusts stock levels for products.


InventoryDao.py
OrderDetailsDao.py
OrderDao.py
Customer.py
Order.py
TechShop > entity > Order.py > Order > get_order_details
from datetime import datetime
from entity.Customer import Customer
class Order:
def __init__(self, order_id=None, customer=None, totalAmount=0, products=None):
self.__order_id = order_id
self.__customer = customer
self.__order_date = datetime.now()
self.__total_amount = totalAmount
self.__products = products if products else []
def calculate_total_amount(self):
self.__total_amount = sum(product.price for product in self.__products)
return self.__total_amount
def get_order_details(self):
pass
def update_order_status(self, status):\
pass
def cancel_order(self):
pass
@property
def order_id(self):
return self.__order_id
@property
def customer(self):
return self.__customer
@customer.setter
def customer(self, customer):
self.__customer = customer
@property
def order_date(self):
return self.__order_date
@order_date.setter
def order_date(self, order_date):
if isinstance(order_date, datetime):
self.__order_date = order_date
else:
raise ValueError("Order date must be a datetime object.")



OrderDetails Class:

Attributes:

• OrderDetailID (int)

• Order (Order) - Use composition to reference the Order to which this detail belongs.

• Product (Product) - Use composition to reference the Product included in the order detail.

• Quantity (int)

Methods:

• CalculateSubtotal() - Calculate the subtotal for this order detail.

• GetOrderDetailInfo(): Retrieves and displays information about this order detail.

• UpdateQuantity(): Allows updating the quantity of the product in this order detail.

• AddDiscount(): Applies a discount to this order detail.

TechShop > entity > OrderDetails.py > OrderDetails

```python

from entity.Order import Order
from exception.exceptions import InvalidDataException


class OrderDetails:
    def __init__(self, order_detail_id=None, order=None, product=None, quantity=0):
        self.__order_detail_id = order_detail_id
        self.__order = order
        self.__product = product
        self.__quantity = quantity

    def calculate_subtotal(self):
        pass

    def get_order_detail_info(self):
        pass

    def update_quantity(self, new_quantity):
        pass

    def add_discount(self, discount):
        pass

    @property
    def order_detail_id(self):
        return self.__order_detail_id

    @property
    def order(self):
        return self.__order

    @order.setter
    def order(self, order):
        self.__order = order

    @property
    def product(self):
        return self.__product

    @product.setter
    def product(self, product):
        self.__product = product

    @property
    def quantity(self):
        return self.__quantity

```

```
48
49      @quantity.setter
50      def quantity(self, quantity):
51          if isinstance(quantity, int) and quantity >= 0:
52              self.__quantity = quantity
53          else:
54              raise InvalidDataException("Quantity must be a non-negative integer.")
55
```

Inventory class:

Attributes:

• InventoryID(int)

• Product (Composition): The product associated with the inventory item.

• QuantityInStock: The quantity of the product currently in stock.

• LastStockUpdate

Methods:

• GetProduct(): A method to retrieve the product associated with this inventory item.

• GetQuantityInStock(): A method to get the current quantity of the product in stock.

• AddToInventory(int quantity): A method to add a specified quantity of the product to the inventory.

• RemoveFromInventory(int quantity): A method to remove a specified quantity of the product from the inventory.

• UpdateStockQuantity(int newQuantity): A method to update the stock quantity to a new value.

• IsProductAvailable(int quantityToCheck): A method to check if a specified quantity of the product is available in the inventory.

• GetInventoryValue(): A method to calculate the total value of the products in the inventory based on their prices and quantities.

• ListLowStockProducts(int threshold): A method to list products with quantities below a specified threshold, indicating low stock.

• ListOutOfStockProducts(): A method to list products that are out of stock.

• ListAllProducts(): A method to list all products in the inventory, along with their quantities.

TechShop > entity > Inventory.py > Inventory

```python
from datetime import datetime
from exception.exceptions import InsufficientStockException, InvalidDataException

class Inventory:
    def __init__(self, inventory_id=None, product=None, quantity_in_stock=0, last_stock_update=None):
        self.__inventory_id = inventory_id
        self.__product = product
        self.__quantity_in_stock = quantity_in_stock if quantity_in_stock>0 else 0
        self.__last_stock_update = last_stock_update if last_stock_update else datetime.now()

    def get_product(self):
        return self.__product

    def get_quantity_in_stock(self):
        return self.__quantity_in_stock

    def add_to_inventory(self, quantity):
        self.__quantity_in_stock += quantity
        self.__last_stock_update = datetime.now()

    def remove_from_inventory(self, quantity):
        if self.__quantity_in_stock >= quantity:
            self.__quantity_in_stock -= quantity
            self.__last_stock_update = datetime.now()
        else:
            raise InsufficientStockException("Insufficient quantity in stock.")

    def update_stock_quantity(self, new_quantity):
        self.__quantity_in_stock = new_quantity
        self.__last_stock_update = datetime.now()

    def is_product_available(self, quantity_to_check):
        return self.__quantity_in_stock >= quantity_to_check

    def get_inventory_value(self):
        return self.__product.price * self.__quantity_in_stock

    def list_low_stock_products(self, threshold):
        if self.__quantity_in_stock < threshold:
            return self.__product

    def list_out_of_stock_products(self):
        if self.__quantity_in_stock == 0:
            return self.__product

    def list_all_products(self):
        return self.__product

```

```python
class Inventory:

    @property
    def inventory_id(self):
        return self.__inventory_id

    @property
    def product(self):
        return self.__product

    @product.setter
    def product(self, product):
        # Add validation logic if needed
        self.__product = product

    @property
    def quantity_in_stock(self):
        return self.__quantity_in_stock

    @quantity_in_stock.setter
    def quantity_in_stock(self, quantity_in_stock):
        if isinstance(quantity_in_stock, int) and quantity_in_stock >= 0:
            self.__quantity_in_stock = quantity_in_stock
        else:
            raise InvalidDataException("Quantity in stock must be a non-negative integer.")

    @property
    def last_stock_update(self):
        return self.__last_stock_update

    @last_stock_update.setter
    def last_stock_update(self, last_stock_update):
        if isinstance(last_stock_update, datetime):
            self.__last_stock_update = last_stock_update
        else:
            raise InvalidDataException("Last stock update must be a datetime object.")

```

Task 4: Composition:

Ensure that the Order and OrderDetail classes correctly use composition to reference Customer and Product objects.

• Orders Class with Composition:

o In the Orders class, we want to establish a composition relationship with the Customers class, indicating that each order is associated with a specific customer.

o In the Orders class, we've added a private attribute customer of type Customers, establishing a composition relationship. The Customer property provides access to the Customers object associated with the order.

```python

from entity.Customer import Customer

class Order:
    def __init__(self, order_id=None, customer: Customer=None, totalAmount=0, products=None):
        self.__order_id = order_id
        self.__customer = customer
        self.__order_date = datetime.now()
        self.__total_amount = totalAmount
        self.__products = products if products else []

```

• OrderDetails Class with Composition:

o Similarly, in the OrderDetails class, we want to establish composition relationships with both the Orders and Products classes to represent the details of each order, including the product being ordered.

o In the OrderDetails class, we've added two private attributes, order and product, of types Orders and Products, respectively, establishing composition relationships. The Order property provides access to the Orders object associated with the order detail, and the Product property provides access to the Products object representing the product in the order detail.

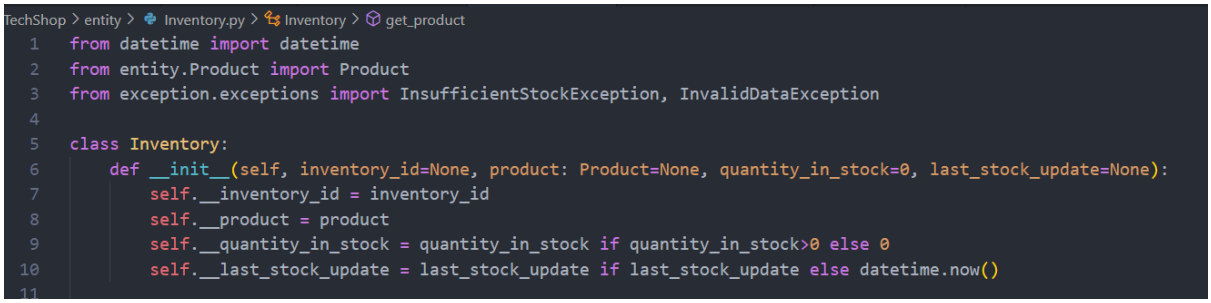
```python
from entity.Order import Order
from entity.Product import Product
from exception.exceptions import InvalidDataException


class OrderDetails:
    def __init__(self, order_detail_id=None, order: Order=None, product: Product=None, quantity=0):
        self.__order_detail_id = order_detail_id
        self.__order = order
        self.__product = product
        self.__quantity = quantity

```

• Customers and Products Classes:

o The Customers and Products classes themselves may not have direct composition relationships with other classes in this scenario. However, they serve as the basis for composition relationships in the Orders and OrderDetails classes, respectively.

• Inventory Class:

o The Inventory class represents the inventory of products available for sale. It can have composition relationships with the Products class to indicate which products are in the inventory.



Task 5: Exceptions handling

• Data Validation:

o Challenge: Validate user inputs and data from external sources (e.g., user registration, order placement).

o Scenario: When a user enters an invalid email address during registration.

o Exception Handling: Throw a custom InvalidDataException with a clear error message.

• Inventory Management:

o Challenge: Handling inventory-related issues, such as selling more products than are in stock.

o Scenario: When processing an order with a quantity that exceeds the available stock.

o Exception Handling: Throw an InsufficientStockException and update the order status accordingly.

• Order Processing:

o Challenge: Ensuring the order details are consistent and complete before processing.

o Scenario: When an order detail lacks a product reference.

o Exception Handling: Throw an IncompleteOrderException with a message explaining the issue.

• Payment Processing:

o Challenge: Handling payment failures or declined transactions.

o Scenario: When processing a payment for an order and the payment is declined.

o Exception Handling: Handle payment-specific exceptions (e.g., PaymentFailedException) and initiate retry or cancellation processes.

• File I/O (e.g., Logging):

o Challenge: Logging errors and events to files or databases.

o Scenario: When an error occurs during data persistence (e.g., writing a log entry).

o Exception Handling: Handle file I/O exceptions (e.g., IOException) and log them appropriately.

• Database Access:

o Challenge: Managing database connections and queries.

o Scenario: When executing a SQL query and the database is offline.

o Exception Handling: Handle database-specific exceptions (e.g., SqlException) and implement connection retries or failover mechanisms.

• Concurrency Control:

o Challenge: Preventing data corruption in multi-user scenarios.

o Scenario: When two users simultaneously attempt to update the same order.

o Exception Handling: Implement optimistic concurrency control and handle ConcurrencyException by notifying users to retry.

• Security and Authentication:

o Challenge: Ensuring secure access and handling unauthorized access attempts.

o Scenario: When a user tries to access sensitive information without proper authentication.

o Exception Handling: Implement custom AuthenticationException and AuthorizationException to handle security-related issues.

TechShop > exception > exceptions.py > ...

```python
# exceptions.py

class InsufficientStockException(Exception):
    def __init__(self, message="Insufficient stock."):
        self.message = message
        super().__init__(self.message)


class InvalidDataException(Exception):
    def __init__(self, message="Invalid data provided."):
        self.message = message
        super().__init__(self.message)

class IncompleteOrderException(Exception):
    def __init__(self, message="Incomplete order details."):
        self.message = message
        super().__init__(self.message)


class PaymentFailedException(Exception):
    def __init__(self, message="Payment failed."):
        self.message = message
        super().__init__(self.message)

```

```python
#product management
class DuplicateProductException(Exception):
    def __init__(self, message="Product already exists."):
        self.message = message
        super().__init__(self.message)

class NotFoundException(Exception):
    def __init__(self, message="Product not found."):
        self.message = message
        super().__init__(self.message)

class ProductHasOrdersException(Exception):
    def __init__(self, message="Product has existing orders and cannot be removed."):
        self.message = message
        super().__init__(self.message)

class FileIOException(Exception):
    def __init__(self, message="Error accessing or performing I/O operation on file."):
        self.message = message
        super().__init__(self.message)
```

```python
class DatabaseOfflineException(Exception):
    def __init__(self, message="Error executing SQL query due to database offline."):
        self.message = message
        super().__init__(self.message)

class ConcurrencyException(Exception):
    def __init__(self, message="Concurrency issue occurred. Please retry."):
        self.message = message
        super().__init__(self.message)

class AuthenticationException(Exception):
    def __init__(self, message="Authentication failed. Invalid username or password."):
        self.message = message
        super().__init__(self.message)

class AuthorizationException(Exception):
    def __init__(self, message="Unauthorized access attempt."):
        self.message = message
        super().__init__(self.message)
```

Task 6: Collections

• Managing Products List:

o Challenge: Maintaining a list of products available for sale (List<Products>).

o Scenario: Adding, updating, and removing products from the list.

o Solution: Implement methods to add, update, and remove products. Handle exceptions for duplicate products, invalid updates, or removal of products with existing orders.

• Managing Orders List:

o Challenge: Maintaining a list of customer orders (List<Orders>).

o Scenario: Adding new orders, updating order statuses, and removing canceled orders.

o Solution: Implement methods to add new orders, update order statuses, and remove canceled orders. Ensure that updates are synchronized with inventory and payment records.

• Sorting Orders by Date:

o Challenge: Sorting orders by order date in ascending or descending order.

o Scenario: Retrieving and displaying orders based on specific date ranges.

o Solution: Use the List<Orders> collection and provide custom sorting methods for order date. Consider implementing SortedList if you need frequent sorting operations.

• Inventory Management with SortedList:

o Challenge: Managing product inventory with a SortedList based on product IDs.

o Scenario: Tracking the quantity in stock for each product and quickly retrieving inventory information.

o Solution: Implement a SortedList<int, Inventory> where keys are product IDs. Ensure that inventory updates are synchronized with product additions and removals.

• Handling Inventory Updates:

o Challenge: Ensuring that inventory is updated correctly when processing orders.

o Scenario: Decrementing product quantities in stock when orders are placed.

o Solution: Implement a method to update inventory quantities when orders are processed. Handle exceptions for insufficient stock.

• Product Search and Retrieval:

o Challenge: Implementing a search functionality to find products based on various criteria (e.g., name, category).

o Scenario: Allowing customers to search for products.

o Solution: Implement custom search methods using LINQ queries on the List<Products> collection. Handle exceptions for invalid search criteria.

• Duplicate Product Handling:

o Challenge: Preventing duplicate products from being added to the list.

o Scenario: When a product with the same name or SKU is added.

o Solution: Implement logic to check for duplicates before adding a product to the list. Raise exceptions or return error messages for duplicates.

• Payment Records List:

o Challenge: Managing a list of payment records for orders (List<PaymentClass>).

o Scenario: Recording and updating payment information for each order.

o Solution: Implement methods to record payments, update payment statuses, and handle payment errors. Ensure that payment records are consistent with order records.

• OrderDetails and Products Relationship:

o Challenge: Managing the relationship between OrderDetails and Products.

o Scenario: Ensuring that order details accurately reflect the products available in the inventory.

o Solution: Implement methods to validate product availability in the inventory before adding order details. Handle exceptions for unavailable products.

TechShop > controller > inventoryManagement.py > InventoryManager > _find_index_by_product_id

```python
from sortedcontainers import SortedList
from datetime import datetime


class InventoryManager:
    def __init__(self):
        self.inventory_list = SortedList()

    def add_inventory(self, inventory):
        self.inventory_list.add((inventory.product.product_id, inventory))

    def remove_inventory(self, product_id):
        index = self._find_index_by_product_id(product_id)
        if index is not None:
            del self.inventory_list[index]

    def update_inventory_quantity(self, product_id, new_quantity):
        index = self._find_index_by_product_id(product_id)
        if index is not None:
            self.inventory_list[index][1].quantity_in_stock = new_quantity

    def get_inventory_by_product_id(self, product_id):
        index = self._find_index_by_product_id(product_id)
        if index is not None:
            return self.inventory_list[index][1]
        return None

    def _find_index_by_product_id(self, product_id):
        for i, (key, inventory) in enumerate(self.inventory_list):
            if key == product_id:
                return i
        return None

```

TechShop > controller > orderManagement.py > OrderManager > update_order_status

```python
from exception.exceptions import NotFoundException


class OrderManager:
    def __init__(self):
        self.__orders = []

    def add_order(self, order):
        self.__orders.append(order)

    def update_order_status(self, order_id, new_status):
        for order in self.__orders:
            if order.order_id == order_id:
                order.update_order_status(new_status)
                return
        raise NotFoundException("Order not found in the list.")

    def remove_canceled_orders(self):
        self.__orders = [order for order in self.__orders if order.status != "canceled"]

    def sort_orders_by_date(self, ascending=True):
        self.__orders.sort(key=lambda order: order.order_date, reverse=not ascending)

    def get_orders_in_date_range(self, start_date, end_date):
        return [order for order in self.__orders if start_date <= order.order_date <= end_date]

    def list_orders(self):
        return self.__orders

```

TechShop > controller > productManagement.py > ProductManager > list_products

```python
from exception.exceptions import DuplicateProductException, ProductHasOrdersException, NotFoundException


class ProductManager:
    def __init__(self):
        self.__products = []

    def add_product(self, product):
        if product not in self.__products:
            self.__products.append(product)
        else:
            raise DuplicateProductException("Product already exists in the list.")

    def update_product(self, product_id, new_product):
        for i, product in enumerate(self.__products):
            if product.product_id == product_id:
                self.__products[i] = new_product
                return
        raise NotFoundException("Product not found in the list.")

    def remove_product(self, product_id):
        for i, product in enumerate(self.__products):
            if product.product_id == product_id:
                # Check if the product has existing orders
                if product.has_orders():
                    raise ProductHasOrdersException("Product cannot be removed as it has existing orders.")
                del self.__products[i]
                return
        raise NotFoundException("Product not found in the list.")

    def list_products(self):
        return self.__products

```

TechShop > controller > paymentManagement.py > PaymentRecordsList > record_payment

```python
class PaymentRecord:
    def __init__(self, order_id, amount, status="Pending"):
        self.order_id = order_id
        self.amount = amount
        self.status = status

class PaymentRecordsList:
    def __init__(self):
        self.payment_records = []

    def record_payment(self, payment_record):
        """
        Record a payment for an order.
        """
        self.payment_records.append(payment_record)

    def update_payment_status(self, order_id, new_status):
        """
        Update payment status for a specific order.
        """
        for payment_record in self.payment_records:
            if payment_record.order_id == order_id:
                payment_record.status = new_status
                return
        # If order_id not found
        raise ValueError(f"No payment record found for order ID: {order_id}")

    def handle_payment_error(self, order_id, error_message):
        """
        Handle payment errors for a specific order.
        """
        for payment_record in self.payment_records:
            if payment_record.order_id == order_id:
                payment_record.status = "Error"
                payment_record.error_message = error_message
                return
        # If order_id not found
        raise ValueError(f"No payment record found for order ID: {order_id}")

```

Task 7: Database Connectivity

• Implement a DatabaseConnector class responsible for establishing a connection to the "TechShopDB" database. This class should include methods for opening, closing, and managing database connections.

TechShop > util > DBConnUtil.py > DBConnUtil

```python
import mysql.connector
from mysql.connector import Error
import time

from util.DBPropertyUtil import DBPropertyUtil

class DBConnUtil:
    __connection = None

    @staticmethod
    def getConnection():
        property_file = r"C:\Users\raaji\OneDrive\Documents\hexaware training\TechShop\db.properties"
        connection_string = DBPropertyUtil().get_connection_string(property_file)
        try:
            DBConnUtil.__connection = mysql.connector.connect(**connection_string)
            print("Connected to MySQL database successfully.")
            return DBConnUtil.__connection
        except mysql.connector.Error as err:
            print("Error connecting to MySQL:", err)
            return None


    @staticmethod
    def close_connection():
        if DBConnUtil.__connection:
            DBConnUtil.__connection.close()
            print("Connection closed.")
            DBConnUtil.__connection = None
```

TechShop > util > DBPropertyUtil.py > DBPropertyUtil > get_connection_string

```python
# util/DBPropertyUtil.py
import configparser

class DBPropertyUtil:
    @staticmethod
    def get_connection_string(property_file):
        config = configparser.ConfigParser()
        config.read(property_file)
        connection_string = {
            'user': config['DATABASE']['user'],
            'host': config['DATABASE']['host'],
            'port': config['DATABASE'].getint('port'),
            'passwd': config['DATABASE']['passwd'],
            'database': config['DATABASE']['database']
        }
        return connection_string

```

• Implement classes for Customers, Products, Orders, OrderDetails, Inventory with properties, constructors, and methods for CRUD (Create, Read, Update, Delete) operations.

```python
from dao.CustomerDao import CustomerDao
from dao.InventoryDao import InventoryDao
from dao.OrderDao import OrderDao
from dao.OrderDetailsDao import OrderDetailsDao
from dao.ProductDao import ProductDao


def main():
    customer_obj = CustomerDao()
    product_obj = ProductDao()
    order_obj = OrderDao()
    orderdetails_obj = OrderDetailsDao()
    inventory_obj = InventoryDao()

    # CRUD OPERATIONS
    print(
        "1. Customers\n2. Products\n3. Orders\n4. OrderDetails\n5. Inventory\n6. Exit\n"
    )
    choice = input("Enter your choice: ")
```

Operations on Customer:

```python
    if choice == "1":
        print(
            "\n1. Insert Customer\n2. Delete Customer\n3. Calculate Total Orders\n4. Get Customer Details\n5. Update Customer Info\n"
        )
        choice_customer = input("Enter your choice: ")
        if choice_customer == "1":
            first_name = input("Enter First Name: ")
            last_name = input("Enter Last Name: ")
            email = input("Enter Email: ")
            phone = input("Enter Phone: ")
            address = input("Enter Address: ")
            customer_obj.insertCustomer(first_name, last_name, email, phone, address)
        elif choice_customer == "2":
            customer_id = input("Enter Customer ID to delete: ")
            customer_obj.deleteCustomer(customer_id)
        elif choice_customer == "3":
            customer_id = input("Enter Customer ID: ")
            customer_obj.calculateTotalOrders(customer_id)
        elif choice_customer == "4":
            customer_id = input("Enter Customer ID: ")
            customer_obj.getCustomerDetails(customer_id)
        elif choice_customer == "5":
            customer_id = input("Enter Customer ID: ")
            email = input("Enter New Email: ")
            phone = input("Enter New Phone: ")
            address = input("Enter New Address: ")
            customer_obj.updateCustomerInfo(customer_id, email, phone, address)
        else:
            print("Invalid choice. Please try again.")
```

Operations on Product:

```
    elif choice == "2":
        print(
            "\n1. Get Product Details\n2. Insert Product\n3. Update Product Info\n4. Check Product Stock\n"
        )
        choice_product = input("Enter your choice: ")
        if choice_product == "1":
            product_id = input("Enter Product ID: ")
            product_obj.get_product_details(product_id)
        elif choice_product == "2":
            product_name = input("Enter Product Name: ")
            description = input("Enter Product Description: ")
            price = input("Enter Product Price: ")
            product_obj.insert_product(product_name, description, price)
        elif choice_product == "3":
            product_id = input("Enter Product ID: ")
            price = input("Enter New Price: ")
            description = input("Enter New Description: ")
            product_obj.update_product_info(product_id, price, description)
        elif choice_product == "4":
            product_id = input("Enter Product ID: ")
            product_obj.is_product_in_stock(product_id)
        else:
            print("Invalid choice. Please try again.")
```

Operations On Order:

```
    elif choice == "3":
        print(
            "\n1. Insert Order\n2. Update Order Status\n3. Cancel Order\n4. Calculate total amount\n5.Get order details"
        )
        choice_order = input("Enter your choice: ")
        if choice_order == "1":
            customer_id = input("Enter Customer ID: ")
            total_amount = input("Enter Total Amount: ")
            order_obj.insert_order(customer_id, total_amount)
        elif choice_order == "2":
            order_id = input("Enter Order ID: ")
            new_status = input("Enter New Status: ")
            order_obj.update_order_status(order_id, new_status)
        elif choice_order == "3":
            order_id = input("Enter Order ID: ")
            order_obj.cancel_order(order_id)
        elif choice_order == "4":
            order_id = input("Enter Order ID: ")
            order_obj.calculate_total_amount(order_id)
        elif choice_order == "5":
            order_id = input("Enter Order ID: ")
            order_obj.get_order_details(order_id)
        else:
            print("Invalid choice. Please try again.")
```

Operations on OrderDetails:

```
    elif choice == "4":
        print(
            "1. Calculate SubTotal\n"
            + "2. Get Order Details\n"
            + "3. Update Quantity\n"
            + "4. Add Discount"
        )
        choice_order_detail = input("Enter your choice: ")
        if choice_order_detail == "1":
            order_detail_id = input("Enter Order Detail ID: ")
            orderdetails_obj.calculate_subtotal(order_detail_id)
        elif choice_order_detail == "2":
            order_detail_id = input("Enter Order Detail ID: ")
            orderdetails_obj.get_order_detail_info(order_detail_id)
        elif choice_order_detail == "3":
            order_detail_id = input("Enter Order Detail ID: ")
            new_quantity = int(input("Enter new quantity: "))
            orderdetails_obj.update_quantity(order_detail_id, new_quantity)
        elif choice_order_detail == "4":
            try:
                order_detail_id = input("Enter Order Detail ID: ")
                totalAfterDiscount = int(input("Enter Discount Percentage: "))
                orderdetails_obj.add_discount(order_detail_id, totalAfterDiscount)
            except ValueError:
                print("Invalid input. Discount must be number.")
        else:
            print("Invalid choice. Please try again.")
```

Operations on Inventory:

```
            print("Invalid choice. Please try again.")

    elif choice == "5":
        print(
            "1. Get Product\n"
            + "2. Get Quantity in Stock\n"
            + "3. Add To Inventory\n"
            + "4. Remove From Inventory\n"
            + "5. Update Stock Quantity\n"
            + "6. Product Available\n"
            + "7. Get Inventory Value\n"
            + "8. List Low Stock Products\n"
            + "9. List Out Of Stock Products\n"
            + "10. List All Products"
        )
        choice_inventory = input("Enter your choice: ")
        if choice_inventory == "1":
            inventory_id = int(input("Enter the inventory id: "))
            product = inventory_obj.get_product(inventory_id)
            print(product)
        elif choice_inventory == "2":
            inventory_id = int(input("Enter the inventory id: "))
            print(inventory_obj.get_quantity_in_stock(inventory_id))
        elif choice_inventory == "3":
            product_id = int(input("Enter the product id: "))
            quantity_in_stock = int(input("Enter the quantity in stock: "))
            last_stock_update = input("Enter the last stock update: ")
            inventory_obj.insert_inventory(
                product_id, quantity_in_stock, last_stock_update
            )
```

```
        elif choice_inventory == "4":
            inventory_id = int(input("Enter the inventory id: "))
            quantity = int(input("Enter the quantity: "))
            inventory_obj.remove_from_inventory(quantity, inventory_id)
        elif choice_inventory == "5":
            inventory_id = int(input("Enter the inventory id: "))
            new_quantity = int(input("Enter the new quantity: "))
            inventory_obj.update_stock_quantity(inventory_id, new_quantity)
```

```
        elif choice_inventory == "6":
            inventory_id = input("Enter Inventory ID: ")
            quantity_to_check = int(input("Enter Quantity to Check: "))
            if inventory_obj.is_product_available(inventory_id, quantity_to_check):
                print("Product is available in inventory.")
            else:
                print("Product is not available in inventory.")
        elif choice_inventory == "7":
            inventory_id = input("Enter Inventory ID: ")
            print(inventory_obj.get_inventory_value(inventory_id))
        elif choice_inventory == "8":
            threshold = int(input("Enter the threshold for low stock: "))
            inventory_obj.list_low_stock_products(threshold)
        elif choice_inventory == "9":
            inventory_obj.list_out_of_stock_products()
        elif choice_inventory == "10":
            inventory_obj.list_all_products()
        else:
            print("Invalid choice. Please try again.")
```

```
                print("Invalid choice. Please try again.")
    elif choice == '6':
        print("Exiting...")
    else:
        print("Invalid choice. Please try again.")

```

Tasks Menu:

```python

from dao.paymentDao import PaymentProcessingSystem
from exception.exceptions import IncompleteOrderException
from dao.CustomerDao import CustomerDao
from dao.InventoryDao import InventoryDao
from dao.OrderDao import OrderDao
from dao.OrderDetailsDao import OrderDetailsDao
from dao.ProductDao import ProductDao



class Menu:
    def __init__(self):
        self.customer_obj = CustomerDao()
        self.product_obj = ProductDao()
        self.order_obj = OrderDao()
        self.orderdetails_obj = OrderDetailsDao()
        self.inventory_obj = InventoryDao()
        self.payment_obj = PaymentProcessingSystem()

    def display_menu(self):
        print("1. Customer Registration")
        print("2. Product Catalog Management")
        print("3. Place Orders")
        print("4. Track Order Status")
        print("5. Inventory Management")
        print("6. Sales Reports")
        print("7. Customer Account Updates")
        print("8. Payment Processing")
        print("9. Product Search and Recommendations")

    def handle_choice(self, choice):
        if choice == "1":
            self.customer_registration()
        elif choice == "2":
            self.product_catalog_management()
        elif choice == "3":
            self.place_orders()
        elif choice == "4":
            self.track_order_status()
        elif choice == "5":
            self.inventory_management()
        elif choice == "6":
            self.sales_reports()
        elif choice == "7":
            self.customer_account_updates()
        elif choice == "8":
            self.payment_processing()
        elif choice == "9":
            self.product_search_and_recommendations()
        else:
            print("Invalid choice")

```

1: Customer Registration

Description: When a new customer registers on the TechShop website, their information (e.g., name, email, phone) needs to be stored in the database.

Task: Implement a registration form and database connectivity to insert new customer records. Ensure proper data validation and error handling for duplicate email addresses.

```
58
59      def customer_registration(self):
60          first_name = input("Enter first name: ")
61          last_name = input("Enter last name: ")
62          email = input("Enter email: ")
63          phone = input("Enter phone number: ")
64          address = input("Enter address: ")
65          self.customer_obj.insertCustomer(first_name, last_name, email, phone, address)
66
```

2: Product Catalog Management

Description: TechShop regularly updates its product catalog with new items and changes in product details (e.g., price, description). These changes need to be reflected in the database.

Task: Create an interface to manage the product catalog. Implement database connectivity to update product information. Handle changes in product details and ensure data consistency.

```
66
67      def product_catalog_management(self):
68          print(
69              "\n1. Get Product Details\n2. Insert Product\n3. Update Product Info\n4. Check Product Stock\n"
70          )
71          choice_product = input("Enter your choice: ")
72          if choice_product == "1":
73              product_id = input("Enter Product ID: ")
74              self.product_obj.get_product_details(product_id)
75          elif choice_product == "2":
76              product_name = input("Enter Product Name: ")
77              description = input("Enter Product Description: ")
78              price = input("Enter Product Price: ")
79              self.product_obj.insert_product(product_name, description, price)
80          elif choice_product == "3":
81              product_id = input("Enter Product ID: ")
82              price = input("Enter New Price: ")
83              description = input("Enter New Description: ")
84              self.product_obj.update_product_info(product_id, price, description)
85          elif choice_product == "4":
86              product_id = input("Enter Product ID: ")
87              self.product_obj.is_product_in_stock(product_id)
88          else:
89              print("Invalid choice. Please try again.")
90
```

3: Placing Customer Orders

Description: Customers browse the product catalog and place orders for products they want to purchase. The orders need to be stored in the database.

Task: Implement an order processing system. Use database connectivity to record customer orders, update product quantities in inventory, and calculate order totals.

```
91
92      def place_orders(self):
93          try:
94              product_id = int(input("Enter Product ID: "))
95              quantity = int(input("Enter quantity: "))
96              customer_id = int(input("Enter Customer ID: "))
97
98              if not self.inventory_obj.is_product_available(product_id, quantity):
99                  raise IncompleteOrderException("Selected product is not available in the required quantity.")
100
101             order_id = self.order_obj.insert_order(customer_id)
102             self.orderdetails_obj.insert_order_detail(order_id, product_id, quantity)
103             self.inventory_obj.remove_from_inventory(product_id, quantity)
104             self.order_obj.calculate_total_amount(order_id)
105             print("Order placed successfully.")
106         except IncompleteOrderException as e:
107             print("Error placing order:", e)
108         except Exception as ex:
109             print("Error placing order:", ex)
```

4: Tracking Order Status

Description: Customers and employees need to track the status of their orders. The order status information is stored in the database.

Task: Develop a feature that allows users to view the status of their orders. Implement database connectivity to retrieve and display order status information.

```
111
112     def track_order_status(self):
113         order_id = input("Enter order ID: ")
114         self.order_obj.get_order_details(order_id)
```

5: Inventory Management

Description: TechShop needs to manage product inventory, including adding new products, updating stock levels, and removing discontinued items.

Task: Create an inventory management system with database connectivity. Implement features for adding new products, updating quantities, and handling discontinued products.

```python
    def inventory_management(self):
        print(
            "1. Get Product\n"
            + "2. Get Quantity in Stock\n"
            + "3. Add To Inventory\n"
            + "4. Remove From Inventory\n"
            + "5. Update Stock Quantity\n"
            + "6. Product Available\n"
            + "7. Get Inventory Value\n"
            + "8. List Low Stock Products\n"
            + "9. List Out Of Stock Products\n"
            + "10. List All Products"
        )
        choice_inventory = input("Enter your choice: ")
        if choice_inventory == "1":
            inventory_id = int(input("Enter the inventory id: "))
            product = self.inventory_obj.get_product(inventory_id)
            print(product)
        elif choice_inventory == "2":
            inventory_id = int(input("Enter the inventory id: "))
            print(self.inventory_obj.get_quantity_in_stock(inventory_id))
        elif choice_inventory == "3":
            product_id = int(input("Enter the product id: "))
            quantity_in_stock = int(input("Enter the quantity in stock: "))
            last_stock_update = input("Enter the last stock update: ")
            self.inventory_obj.insert_inventory(
                product_id, quantity_in_stock, last_stock_update
            )
        elif choice_inventory == "4":
            product_id = int(input("Enter the product id: "))
            quantity = int(input("Enter the quantity: "))
            self.inventory_obj.remove_from_inventory(product_id, quantity)
        elif choice_inventory == "5":
            inventory_id = int(input("Enter the inventory id: "))
            new_quantity = int(input("Enter the new quantity: "))
            self.inventory_obj.update_stock_quantity(inventory_id, new_quantity)
```

```python
        elif choice_inventory == "6":
            inventory_id = input("Enter Inventory ID: ")
            quantity_to_check = int(input("Enter Quantity to Check: "))
            if self.inventory_obj.is_product_available(inventory_id, quantity_to_check):
                print("Product is available in inventory.")
            else:
                print("Product is not available in inventory.")
        elif choice_inventory == "7":
            inventory_id = input("Enter Inventory ID: ")
            print(self.inventory_obj.get_inventory_value(inventory_id))
        elif choice_inventory == "8":
            threshold = int(input("Enter the threshold for low stock: "))
            self.inventory_obj.list_low_stock_products(threshold)
        elif choice_inventory == "9":
            self.inventory_obj.list_out_of_stock_products()
        elif choice_inventory == "10":
            self.inventory_obj.list_all_products()
        else:
            print("Invalid choice. Please try again.")

```

6: Sales Reporting

Description: TechShop management requires sales reports for business analysis. The sales data is stored in the database.

Task: Design and implement a reporting system that retrieves sales data from the database and generates reports based on specified criteria.

```python
def fetch_sales_data(self, start_date, end_date):
    try:
        connection = DBConnUtil.getConnection()
        cursor = connection.cursor()
        sql_query = """
                    SELECT o.OrderID, o.OrderDate, o.TotalAmount, od.ProductID, od.Quantity
                    FROM orders o
                    INNER JOIN orderdetails od ON o.OrderID = od.OrderID
                    WHERE o.OrderDate BETWEEN %s AND %s
                    """
        cursor.execute(sql_query, (start_date, end_date))
        sales_data = cursor.fetchall()

        # Fetch column names
        column_names = [i[0] for i in cursor.description]

        # Print column names
        print("\t".join(column_names))

        # Print data with column names
        for row in sales_data:
            print("\t".join(str(val) for val in row))

    except mysql.connector.Error as e:
        print("Error fetching sales data:", e)
    finally:
        connection.close()
```

```python
    def sales_reports(self):
        start_date = input("Enter Start Date: ")
        end_date = input("Enter end date: ")
        self.order_obj.fetch_sales_data(start_date=start_date, end_date=end_date)
```

7: Customer Account Updates

Description: Customers may need to update their account information, such as changing their email address or phone number.

Task: Implement a user profile management feature with database connectivity to allow customers to update their account details. Ensure data validation and integrity.

```python
    def customer_account_updates(self):
        customer_id = input("Enter Customer ID: ")
        email = input("Enter New Email: ")
        phone = input("Enter New Phone: ")
        address = input("Enter New Address: ")
        self.customer_obj.updateCustomerInfo(customer_id, email, phone, address)
```

8: Payment Processing

Description: When customers make payments for their orders, the payment details (e.g., payment method, amount) must be recorded in the database.

Task: Develop a payment processing system that interacts with the database to record payment

transactions, validate payment information, and handle errors.

```
class PaymentProcessingSystem:

    def record_payment(self, order_id, payment_method, amount):
        try:
            connection = DBConnUtil.getConnection()
            cursor = connection.cursor()
            sql_query = """
                        INSERT INTO payments (OrderID, PaymentMethod, Amount)
                        VALUES (%s, %s, %s)
                        """
            cursor.execute(sql_query, (order_id, payment_method, amount))
            connection.commit()

            print("Payment recorded successfully")
        except Error as e:
            print("Error recording payment:", e)
        finally:
            connection.close()
```

```
    def payment_processing(self):
        order_id = int(input("Enter Order ID: "))
        payment_method = input("Enter Payment Method: ")
        amount = input("Enter amount: ")
        self.payment_obj.record_payment(order_id=order_id, payment_method=payment_method, amount=amount)

```

9: Product Search and Recommendations

Description: Customers should be able to search for products based on various criteria (e.g., name, category) and receive product recommendations.

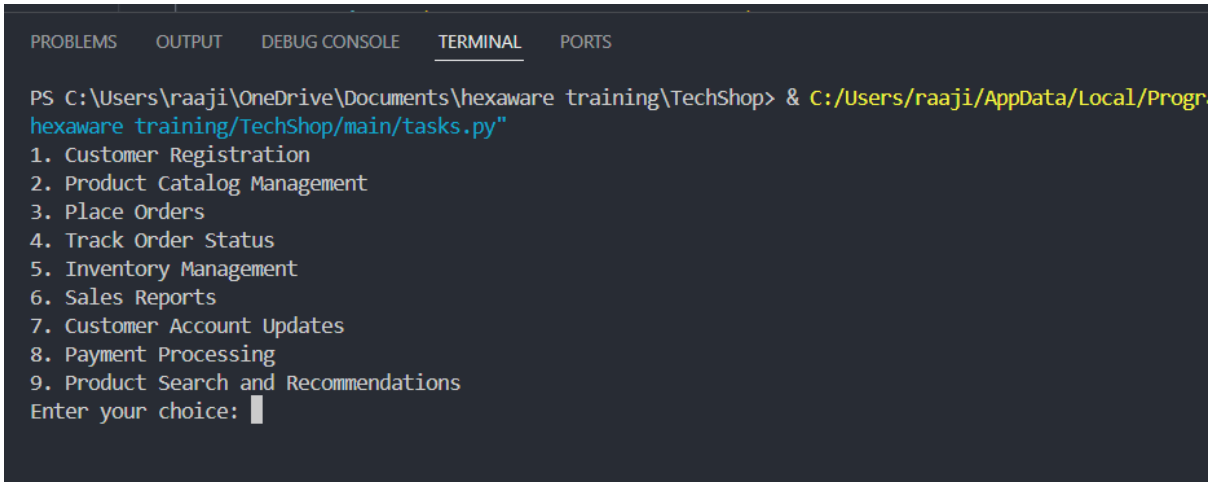
Task: Implement a product search and recommendation engine that uses database connectivity to retrieve relevant product information

```
92
93      def search_product_by_name(self, keyword):
94          try:
95              sql_query = """
96                  SELECT ProductID, ProductName, Description, Price
97                  FROM Products
98                  WHERE ProductName LIKE %s
99              """
100             self.cursor.execute(sql_query, ('%' + keyword + '%',))
101             products = self.cursor.fetchall()
102             if products:
103                 print("Search Results:")
104                 for product in products:
105                     print(product)
106             else:
107                 print("No products found matching the search criteria.")
108         except Exception as e:
109             print("Error searching for products:", e)
```

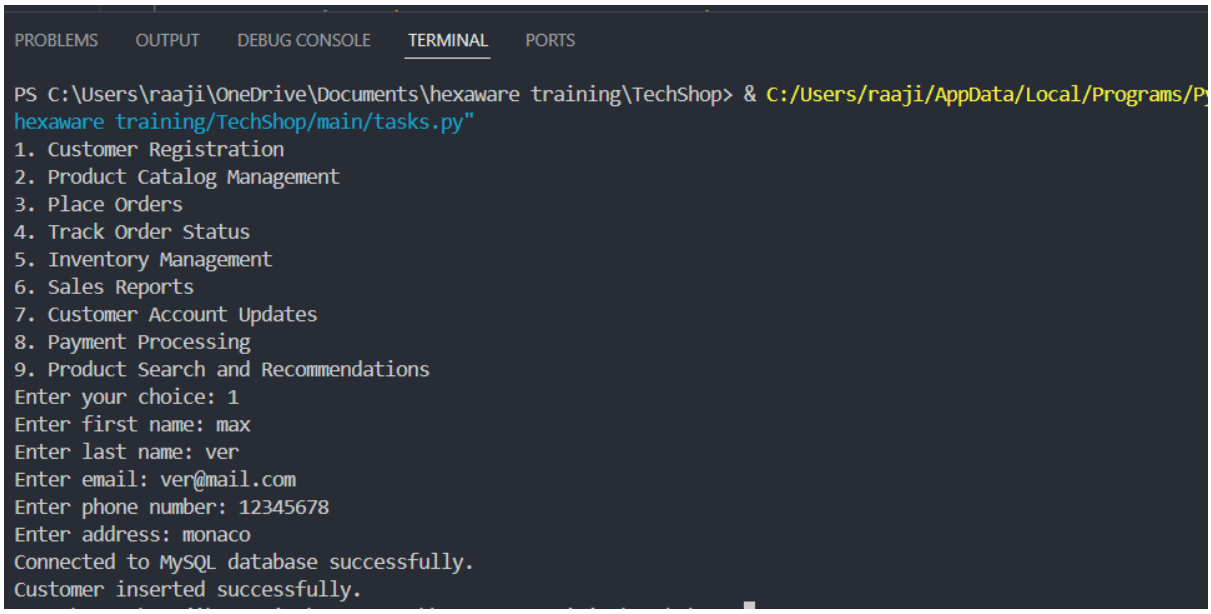
```
114
115     def get_product_recommendations(self, category):
116         try:
117             # Get product recommendations based on the specified category
118             connection = DBConnUtil.getConnection()
119             cursor = connection.cursor()
120             sql_query = """
121                 SELECT ProductID, ProductName, Description, Price
122                 FROM Products
123                 WHERE Category = %s
124                 LIMIT 5
125             """
126             cursor.execute(sql_query, (category,))
127             recommendations = cursor.fetchall()
128             if recommendations:
129                 print("Product Recommendations:")
130                 for product in recommendations:
131                     print(product)
132             else:
133                 print("No recommendations found for the specified category.")
134         except Exception as e:
135             print("Error getting product recommendations:", e)
136         finally:
137             connection.close()
138
```

```python
    def product_search_and_recommendations(self):
        print("1. Search for a product\n2. Get Recommendations\n")
        choice = input("Enter your choice: ")
        if choice == '1':
            keyword = input("Search for a product: ")
            self.product_obj.search_product_by_name(keyword=keyword)
        elif choice == '2':
            category = input("Enter a category: ")
            self.product_obj.get_product_recommendations(category=category)
        else:
            print("Invalid choice. Try again")
```

Menu Output:



Customer Registration:



Product Catalog Management:

```
hexaware training/TechShop/main/tasks.py"ware training\TechShop>
1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
7. Customer Account Updates
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 2

1. Get Product Details
2. Insert Product
3. Update Product Info
4. Check Product Stock

Enter your choice: 1
Enter Product ID: 1
Connected to MySQL database successfully.
Product ID: 1
Product Name: HP Laptop
Description: High performance laptop
Price: 54999.99
```

Place Orders:

```
PS C:\Users\raaji\OneDrive\Documents\hexaware training\TechShop> & C:/Users/raaji/AppData/Local/Programs/P
hexaware training/TechShop/main/tasks.py"
1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 3
Enter Product ID: 3
Enter quantity: 1
Enter Customer ID: 20
Order inserted successfully.
Order detail inserted successfully.
Quantity removed from inventory successfully.
Total Amount calculated and updated successfully.
Order placed successfully.                                  & C:/Users/raaji/AppData/Local/Programs/P
```

Track Order Status:

```
No details found for Order ID: 10                           & C:/Users/raaji/AppData/Local/Program
hexaware training/TechShop/main/tasks.py"ware training\TechShop>
1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
7. Customer Account Updates
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 4
Enter order ID: 1
Order Details for Order ID: 1
Product: HP Laptop, Quantity: 1, Status: shipped
```

Inventory Management:

```
Enter your choice: 5
1. Get Product
2. Get Quantity in Stock
3. Add To Inventory
4. Remove From Inventory
5. Update Stock Quantity
6. Product Available
7. Get Inventory Value
8. List Low Stock Products
9. List Out Of Stock Products
10. List All Products
Enter your choice: 5
Enter the inventory id: 1
Enter the new quantity: 10
Stock quantity updated successfully.
```

Sales Report:

```
hexaware training/TechShop/main/tasks.py"ware training\TechShop>
1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
7. Customer Account Updates
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 6
Enter Start Date: 2024-01-01
Enter end date: 2024-12-12
OrderID OrderDate       TotalAmount     ProductID       Quantity
1       2024-04-01      54999.99        1       1
2       2024-04-02      329999.90       2       10
3       2024-04-03      16499.99        3       1
4       2024-04-04      10999.99        4       1
5       2024-04-05      8799.99 5       1
6       2024-04-06      38499.99        6       1
7       2024-04-07      60499.99        7       1
9       2024-04-09      10999.99        9       1
PS C:\Users\raaji\OneDrive\Documents\hexaware training\TechShop>
```

Customer Account Updates:

```
hexaware training/TechShop/main/tasks.py"ware training\TechShop>
1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
7. Customer Account Updates
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 7
Enter Customer ID: 1
Enter New Email: alonso@mail.com
Enter New Phone: 12345678
Enter New Address: italy
Customer information updated successfully.
PS C:\Users\raaji\OneDrive\Documents\hexaware training\TechShop>
```

Payment Processing:

```
hexaware training/TechShop/main/tasks.py"
1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
7. Customer Account Updates
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 8
Enter Order ID: 1
Enter Payment Method: UPI
Enter amount: 54999
Payment recorded successfully
```

Product Search:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
7. Customer Account Updates
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 9
1. Search for a product
2. Get Recommendations

Enter your choice: 1
Search for a product: laptop
Search Results:
(1, 'HP Laptop', 'High performance laptop', Decimal('54999.99'))
(11, 'Dell Laptop', 'High performance laptop', Decimal('49999.99'))
PS C:\Users\raaji\OneDrive\Documents\hexaware training\TechShop>
```

Product Recommendations:

```
hexaware training/TechShop/main/tasks.py"ware training\TechShop>
1. Customer Registration
2. Product Catalog Management
3. Place Orders
4. Track Order Status
5. Inventory Management
6. Sales Reports
7. Customer Account Updates
8. Payment Processing
9. Product Search and Recommendations
Enter your choice: 9
1. Search for a product
2. Get Recommendations

Enter your choice: 2
Enter a category: mobile
Product Recommendations:
(2, 'Smartphone', 'Latest smartphone model', Decimal('32999.99'))
(12, 'mobile', 'gaming mobile', Decimal('90000.00'))
```